Model-based service creation in the Friends project

W.B. Teeuw
Telematics Institute
Enschede, The Netherlands
teeuw@telin.nl

D.A.C. Quartel
CTIT-University of Twente
Enschede, The Netherlands
quartel@cs.utwente.nl

Abstract—This paper presents a model-based approach to service creation. We observe that the complexity of software services increases. To manage this complexity, and to quickly create specific services in an efficient and cost-effective way upon user request, models are used, going towards 'higher-level' programming. A service creation environment is developed that supports the modelling of services at successive abstraction levels, the analysis of service models, their actual implementation, and the testing and deployment of service implementations. Services are assumed to be developed from existing or newly developed software components. Components are modelled by describing their external behaviour, rather than their interface(s) only. This provides additional design information facilitating a systematic approach to service creation. This paper shows how we model services and their constituent components, and how we use these models.

Key words—Service creation, component-based design, modelling, systems architecting, behaviour specification.

1. Introduction

In the Friends project, a middleware platform has been built to support the development, deployment and management of distributed services [FRIENDS]. An important new feature of this platform is the integrated support for service creation, providing a so-called *service creation environment*. Upon user demand, the service creation environment enables a service developer to design and implement the requested service in an efficient and cost-effective way. A service is assumed to be composed from software components that conform to the middleware platform's underlying component architecture. The service creation environment promotes the re-use of existing components and supports the development of new components if needed. To enable rapid service development, the FRIENDS platform includes generic components supporting access control, authentication (PKI), accounting, performance monitoring, and QoS management, and multimedia components supporting Audio/Video streaming and CSCW services.

A model-based approach underlies the service creation environment. We observe that the complexity of software services increases and the allowed development time of services becomes shorter. A model-based approach helps to manage this complexity, to structure and facilitate service development —going towards 'higher-level' programming— and to validate services at design time. An important characteristic of our approach is to model the complete external behaviour of a component, defining both the operations that can be invoked on its interfaces and the operations invoked by this component on interfaces of other components, as well as the relationships between these operations and their parameters.

The purpose of this paper is twofold: to describe our model-based approach and to present the tool architecture of the service creation environment supporting our approach. A key element of this tool architecture is the modelling tool Testbed Studio [EJO+99, FrJa98]. This paper describes how we use this tool to specify and analyse the behaviour of services and components, and how this tool is integrated in the service creation environment. This paper is further structured as follows: section 2 describes the Friends middleware platform, section 3 motivates and introduces our model-based approach to service creation, section 4 presents the tool architecture of the service creation environment and illustrates the use of the tools identified in this architecture with examples, and section 5 presents our conclusions and future work.

2. The Friends middleware platform

Middleware platforms shield the heterogeneity of underlying operating systems and networks and provide distribution transparencies to applications [Bern96]. Building such a platform, Friends does not start from scratch but uses the results of the Mesh project [BBVD99]. Mesh built a CSCW platform, which complies with the TINA service architecture. Among its features are network independence, user and session mobility, and the support for multimedia stream bindings. In Friends, the functionality of this middleware platform is extended and improved to support not only CSCW applications, but also applications in the area of E-commerce, entertainment and content engineering.

Characteristic for the Friends platform is an *integrated* approach to support service users, providers, and developers (see Figure 1). Given an arbitrary application, e.g., a CSCW environment for project co-operation, an electronic 'Game Hall' for entertainment, or an e-commerce environment for B2B transactions, FRIENDS offers services to all three categories of stake-holders. Video-conferencing, chat, messaging and application sharing are typical examples of functionality that supports the end-users. Service management, accounting and billing functionality typically supports the service providers. The functionality provided by the service creation environment to support service developers is the subject of this paper.

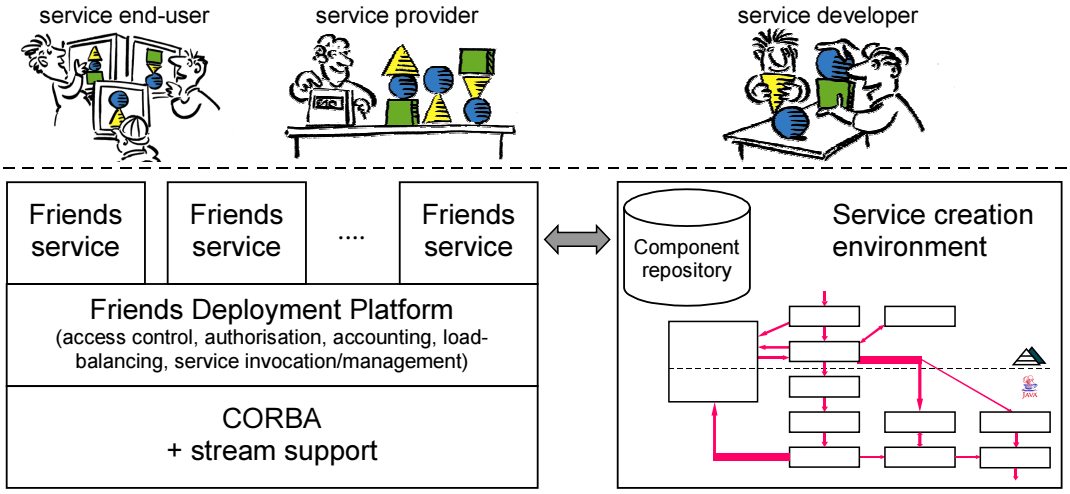

Figure 1. Friends integrated platform approach.

An Internet Service Provider (ISP) typically exploits the Friends Deployment Platform. Alternatively, a retailer (E-tailer) or Application Service Provider (ASP) may exploit the Friends Platform as a whole, including the Friends services on top of it. An Independent Software vendor (ISV) typically exploits the service creation environment. In principle, Friends services may be provided by third parties because they only need to use the deployment platform APIs.

The Friends Deployment Platform is based on component software [Szyp98]. Its underlying Distributed Software Component (DSC) *architecture* defines the minimal rules and constraints a component should adhere to, in order to achieve some minimal level of interoperability with other components [BaBa98]. DSC was developed as a proprietary component architecture because at that time no alternative existed. Recently the CORBA Component Model emerged [CCM99], and DSC will be migrated to this component architecture. The DSC *framework* implements the DSC architecture. The framework defines amongst others the representation in which the component stores its implementation and its specification. The Friends Deployment Platform has been implemented in Java and uses CORBA to support the interaction between distributed components.

3. Service creation: the need for a model-based approach

We define a *service* as the external observable behaviour of a system, which consists of the interactions between the system and its environment and the relationships between these

interactions. Typical examples of systems are applications and software components, with operations invoked on/by components being examples of interactions. Applications are composed from one or more software components, where each component is either an atomic component or a compound component. Consequently, the service provided by an application is composed from the services provided by its constituent components. The term *Friends service* is used to denote the service provided by an application that is deployed by a service provider on the Friends deployment platform.

3.1 Limitations of component-based design

One of the promises of component-based design is the quick introduction of new applications through re-use of software components. Ideally, an application can be developed by selecting available components and composing them such that the requested service is provided. Still many problems need to be solved, however, to realize this ideal picture, most of them originating from a need for methodological support. We mention some of these problems.

Finding components. Components are stored in libraries or repositories. In case repositories are not organised according to proper *classification criteria* it is difficult to find the software one is looking for. Furthermore, standard rules for documenting components in repositories are needed in order to support intelligent *search methods*.

Understanding components. Components are commonly described in terms of interface definitions, one for each component interface, using an Interface Definition Language (IDL). These interface definitions generally describe only the signatures (names and types) of the component's properties, operations and operation arguments. A list of operation's signatures, however, does not completely define how the component behaves. As a consequence, different implementations of the same interface definition may show different behaviours.

Architectural design. Designing a proper composition of components that provides the requested service is a non-trivial task, especially for more complex services. Starting from a specification of the requested service, multiple designs of the requested service at successively, related abstraction levels may precede the final design in order to manage the design complexity. Incorporation of available components in the early design steps shortens the entire design process. Therefore, methodological support is needed that combines a top-down design approach, including specification techniques and decomposition guidelines (e.g., design patterns), with bottom-up knowledge about available software components.

Correctness. Since IDL specifications incompletely define a component's behaviour, it is difficult, actually impossible, to assess the correctness of this component. At best, test runs obtained by executing a component implementation can be used to determine whether the actual (executed) behaviour corresponds to the "expected" behaviour. Many errors found in component implementations can however already be detected during the specification and design phases. Since adaptations in the implementation phase can be very expensive, service specifications and designs should be analysed and validated before.

3.2 Model-based service creation

To tackle the problems as described in the previous section, we propose a model-based approach to service creation [QuSF99]. Modelling is an essential activity when dealing with the inherent complexity of systems. We observe that services become progressively larger and more complex. Models help us to understand services by representing only their *essentials*, i.e., by eliminating everything we consider irrelevant to what we want to consider.

A model-based approach supports *architectural design*, through structuring the service creation process into multiple design steps. Starting from an abstract specification of the requested service, each step delivers a more refined design modelling those service characteristics that are considered relevant at the respective abstraction (or refinement) levels. In this way, separation of concerns is achieved to manage the design complexity. Use

of bottom-up knowledge should guide the design (or refinement) steps to optimise re-use of software components and to obtain the final design in a fast and effective way.

Modelling services (applications) at successively, related abstraction levels, enables validation of the *correctness* of designs. Each successive design step should produce a design that conforms to the designs defined in previous steps. Techniques can be developed to verify this conformance relation (semi-)automatically. The modelling of services also makes the testing of implementations more meaningful, by providing some reference against which the validity of test runs can be checked.

Modelling helps designers to *understand* services and components by representing only their essentials, i.e., the characteristics considered relevant at a certain abstraction level. For example, a service specification should define completely and unambiguously the external behaviour of an application or component. In this way, for example, a service designer can determine the composite behaviour of a certain composition of components and thus decide whether this composition provides the requested behaviour.

Modelling the external behaviour of components can be extended with information about the environment (or context) in which the component can be used and the problems it solves. This resembles the idea of a design pattern [GHJV95]: a specification in terms of a problem, a context, and the (partial) solution as provided by the component. Modelling components this way not only supports a proper *understanding* of components, but also supports the process of *finding components* in a problem-oriented way. For example, it facilitates a problem-oriented categorisation of components and the identification of keywords to be used by component search engines.

3.3 Behaviour specification

To support component-based design, it is generally recognised that the external (operational) behaviour of a component should be defined and, furthermore, be added to the component [Szyp98]. A modelling language is needed that allows one to express the relevant behaviours characteristics of components, with a formal semantics to support analysis and validation. Furthermore, tool support should be available to facilitate the use of such a modelling language. We investigated the following alternatives.

Java. In the Friends project, Java is used as implementation language. Java visual assemblers [Diak99], like Inprise JBuilder, Symantec Visual Cafe, IBM Visual Age, or NetBeans, have simple component assembly features. In addition, extensions to the Java language may be defined to specify the behaviour of components. A straightforward way seems to add pre- and post-conditions to the operation signatures. Some examples already exist, like Biscotti, an extension of Java in which method specifications are extended with (Eiffel-style) preconditions, postconditions and invariants [CiRo99]. The specification of preconditions, postconditions and invariants can be used for run-time checks performed by either the caller (client) or the called (server) component [BJPW99]. However, the combination of Java visual assemblers with Java language extensions does do not support the *abstract* modelling of the behaviours of individual or compositions of software components. Therefore, they are not suited for model-based service creation, but are solely used at implementation level.

UML. UML-based tools support notations for describing different aspects of the structure and behaviour of software, and often suggest methodologies for applying these notations throughout the software development process [UML]. However, UML is not very suited for modelling and relating the behaviour of components at successive (higher) abstraction levels. Furthermore, UML lacks a formal semantics that supports analysis and validation of behaviour specifications.

Formal Description Techniques (FDTs). To support the modelling and design-time analysis of components at higher abstraction levels, other specification techniques need to be considered. In the last decades, many (formal) specification languages have been developed supporting varying conceptual models. FDTs that support an asynchronous interaction model,

like Estelle or SDL, are particularly suited for representing designs at the lower abstraction levels [QFS+97]. To support different and related abstraction levels or constraint-oriented specification styles, a synchronous interaction model is more suited, like in LOTOS [BoLV95] or AMBER [EJO+99].

The use of formal specification languages in distributed systems design is however not widely accepted. The primary reason for this seems to be the required effort to specify designs, which is considerable compared to the effort needed to actually implement the design. We believe the use of a specification (modelling) technique does pay off in terms of improved efficiency of the service creation process and quality of the resulting designs, if providing an integrated tool environment supporting the specification, systematic design, analysis, verification and testing of services.

Amber and Testbed Studio. In the Friends project we have chosen AMBER [EJO+99] as the modelling language, which is supported by an integrated tool environment, called Testbed Studio [FrJa98]. Strong points of AMBER are its expressiveness (allowing those behaviour characteristics to be modelled we consider relevant for service creation), its underlying formalisms (enabling different types of analysis), and its graphical representation. Testbed Studio supports the editing of AMBER specifications, including syntax and semantics checking, and adds a number of analysis tools, such as step-wise simulation, quantitative analysis, integrated use of the model checker SPIN [Holz97], and several kinds of generated views on a model. An additional important factor favouring the choice of AMBER is that the Telematics Institute developed Testbed Studio in the Testbed project [FrJa98]. Therefore, as opposed to other tools, we are able to influence the further development of Testbed Studio.

4. Architecture of the Friends service creation environment

Figure 2 depicts the tool architecture of the Friends service creation environment, identifying the tools supported by this environment and their relationships. The upper part of the figure shows the tools related to the specification and design of services using the modelling language AMBER. The lower part of the figure shows the tools related to the implementation of components in Java and their assembly into deployable services.

This section further explains the tools identified in Figure 2, including their development status. The tools are divided into the following categories: design tools, implementation tools, analysis tools and deployment tools.

4.1 Design tools

Design tools assist the service developer in modelling and designing services (applications) at different abstraction levels. The following design tools are distinguished: modelling tool, modelling library and method support.

4.1.1 Modelling tool

Testbed Studio is used to edit AMBER models. To support the service developer in the modelling process, a mapping is defined between concepts from the DSC component architecture and AMBER concepts. This mapping describes how a DSC concept, such as, e.g., an interface, operation or event, can be modelled in AMBER [QuTe00].

An Amber model consists of three sub-models: (i) an *actor model*, which defines the actors, involved, e.g., software components or functional application parts for which the assignment to components is yet undefined, and how they are interconnected, (ii) a *behaviour model*, which defines the behaviour (functionality) of each actor, and (iii) an *item model*, which defines the objects or data being manipulated by the actors through their behaviour.

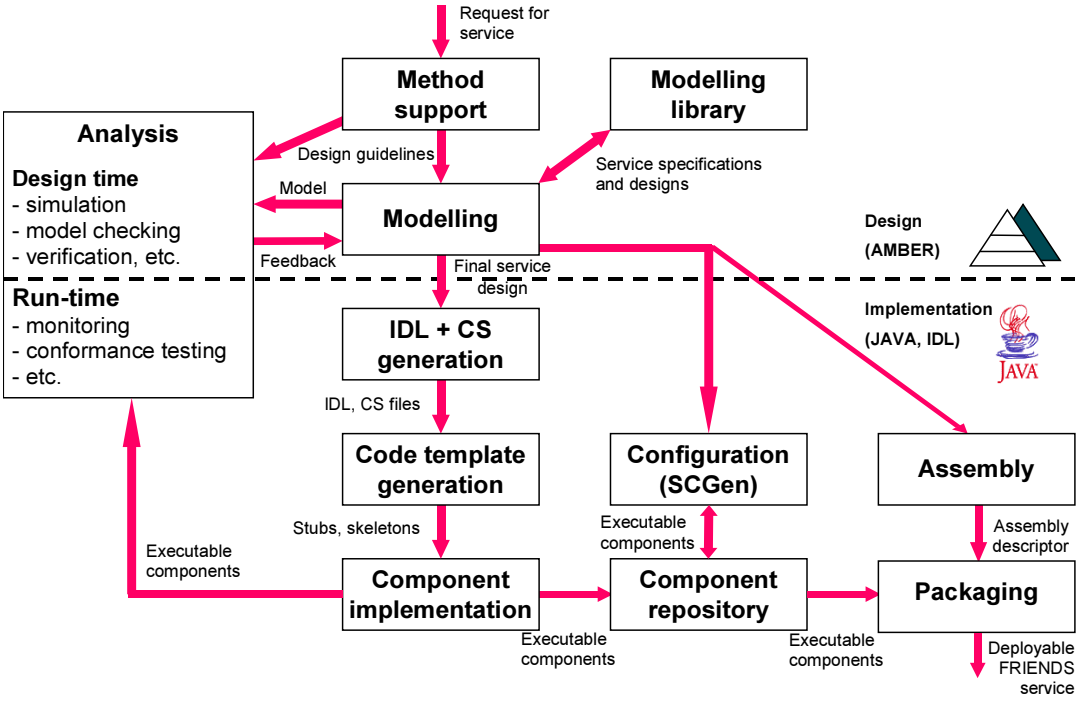

Figure 2. Architecture of the Friends service creation environment.

Actor model. Figure 3 depicts an actor model representing a design of the Friends Shared White Board service. In AMBER, components are modelled by actors (represented as octagons) and interfaces are modelled by interaction points (represented as ovals) [QuTe00].

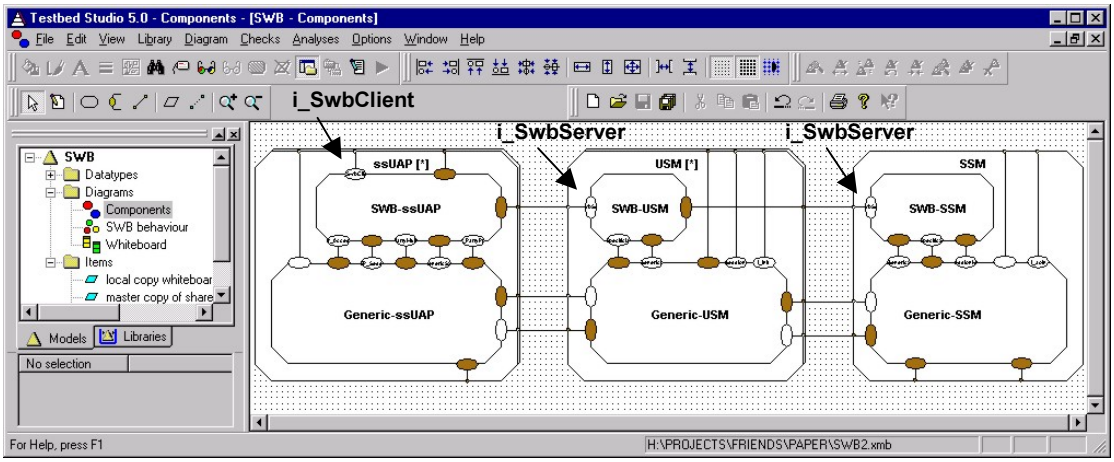

Figure 3. SWB actor model

The design of a Friends service should conform to the TINA service layer architecture. A service is decomposed into a service session User Application (ssUAP) function, which presents the service to the end-user, a Service Session Manager (SSM) function, which maintains the global view of a service session in terms of the parties, stream bindings and resources involved, and a User Session Manager (USM) function, which serves as a security guard between the ssUAP and SSM function to guarantee controlled access to the SSM. Each of these functions is assigned to a separate software component. The entire service is implemented by one instance of the SSM component, and multiple instances of the ssUAP and USM components, one per end-user. Each ssUAP, USM and SSM component is further decomposed into a *generic sub-component*, providing generic management functionality, such as starting and deleting service sessions and adding participants and streambindings to a service session, and *a service specific sub-component*. Consequently, a Friends service is built by extending the generic ssUAP, SSM and USM sub-components with service specific functionality, assigned to the SWB-ssUAP, SWB-SSM and SWB-USM [VWBB99].

Behaviour model. Figure 4 depicts the behaviour models of the SWB-ssUAP, SWB-SSM and SWB-USM, which only consider the i_SwbClient and i_SwbServer interface and abstract from generic service functionality.

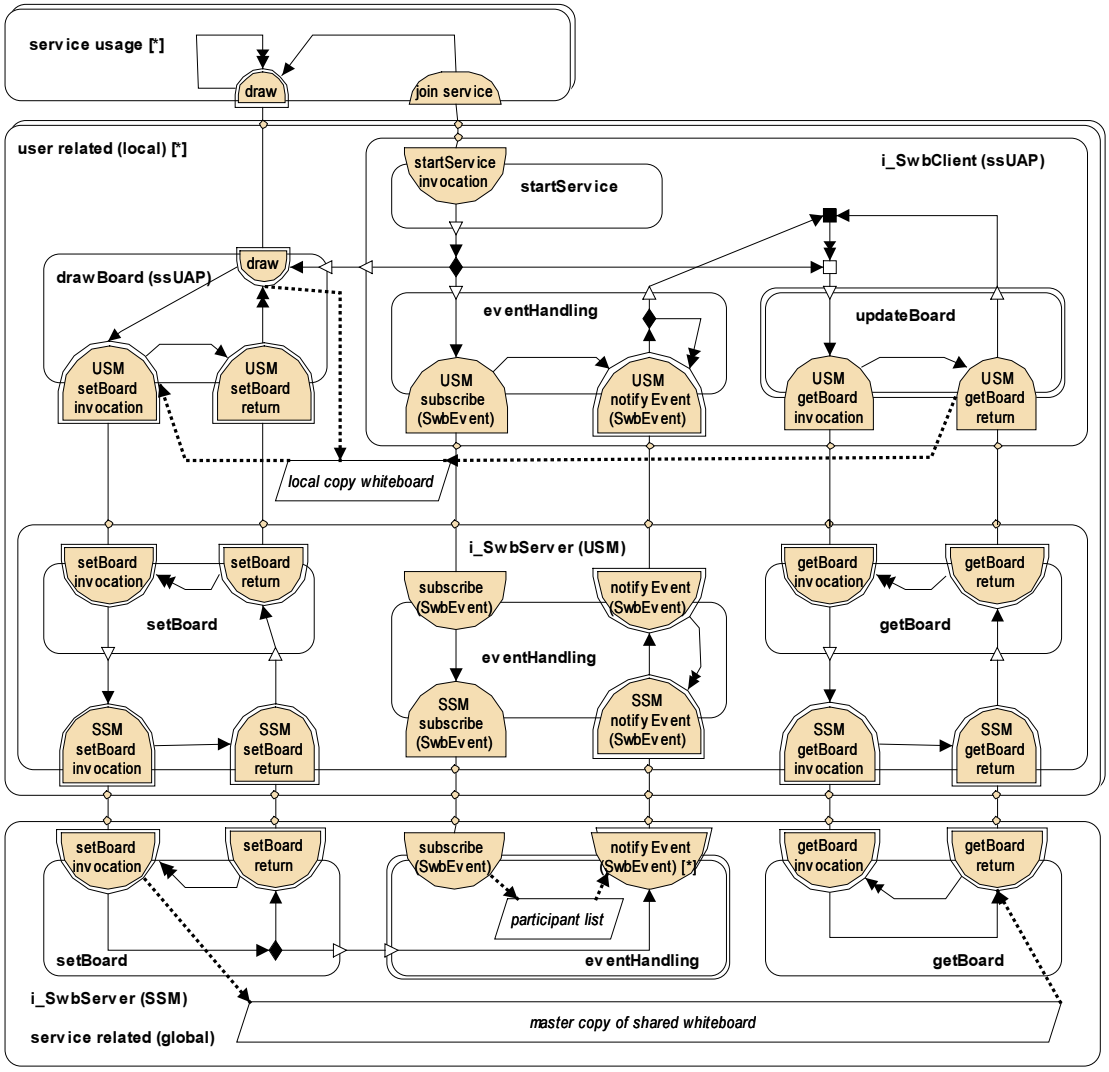

Figure 4. SWB behaviour model

Interface i_SwbClient provides a single operation startService(). This operation is called upon initialisation of the ssUAP. After startService is called:

- the ssUAP updates the whiteboard via operation (USM) getBoard, which is modelled by a sequence of two interactions (USM) getBoard invocation and (USM) getBoard return, which represent the operation invocation and the returning of its result, respectively;
- the ssUAP registers itself via operation (USM) subscribe to listen to so-called SwbEvents, which indicate an update of the shared whiteboard due to a drawing activity by another user. Subsequently, in an end-less loop, the interface listens to shared whiteboard update events (fired by the USM). On receiving a SwbEvent via operation (USM) notifyEvent, the ssUAP updates the whiteboard via operation USM getBoard;
- some drawing functionality is enabled. This functionality is part of the SWB-ssUAP behaviour, but is not part of the i_SwbClient interface. The drawing functionality is modelled as the repeated execution of a draw operation. Each draw operation is followed by updating the shared whiteboard through operation (USM) setBoard.

Interface i_SwbServer provided by the USM merely forwards getBoard and setBoard invocations of the ssUAP to the SSM, as well as forwards the return messages in opposite direction. Similarly, the USM forwards events between ssUAP and SSM.

Interface i_SwbServer provided by the SSM returns the shared whiteboard status upon a getBoard invocation or updates the status upon a setBoard invocation. The shared whiteboard status is represented by the item master copy of shared whiteboard. In case the status is updated an event is fired to notify all USMs involved (the replication of operation notifyEvent models that several listeners are notified). The firing and accepting events is modelled as an announcement. Iteration (loops) are used to model that operations can be invoked repeatedly.

Component composition. The behaviour model of each component in Figure 4 not only models the operations that can be invoked on its interface, which is denoted as the *invoked interface* (facet), but also the invocation of operations on the interface of another component, which is denoted as *the invoking interface* (receptacle). Components are graphically composed in Amber by connecting the interaction points modelling the corresponding invoking (colored grey) and invoked (coloured white) interfaces. The correctness of such a composition, i.e., whether operations of the invoking and invoked interface match, are defined in terms of (static) semantics checks on interactions and interaction points in Amber.

4.1.2 Method support

Having only a modelling tool (editor), i.e., a language and tool support for it, is not enough to create services. One also needs methodological support that tells us how to use the concepts of the language to build services. Methodological support is captured into guidelines and heuristics, service architectures (high-level software and its application to problems), and design methods.

The Friends service creation process consists of the following phases. Starting point is the user's request for service.

1. *Specification* In this phase, a specification of the FRIENDS service is made. The main activity is scoping: deciding on what is "inside" or "outside" the service. The service specification defines the external observable application behaviour and should not define internal behaviour aspects. A well-specified service is one that is both desirable (client satisfaction) and feasible (builder feasibility). This phase also involves the determination of service objectives, service requirements, and use-cases.

2. *Design* This phase considers the Friends service (application) from the internal perspective, by decomposing the service into multiple related functional parts. This decomposition step may be applied recursively to the identified parts, resulting into designs at successive abstraction levels. It is an episodic process of grouping versus separation of related solutions and problems, until a design is obtained that allows a direct mapping of the identified functional past onto existing or implementable software components. This design is called the *final design*. A well-designed service conforms to proven architectures or patterns, and maximises the re-use of existing software components. This phase includes issues like the use of patterns, searching for re-usable components, and design-time validation (see section 4.2.1).

3. *Implementation* The components needed for service implementation either exist or need to be developed. In the former case they are retrieved from the component repository and configured. In the latter case, interface definitions (IDL) and component structure (CS) are derived from the final design, and the components are implemented. A well-implemented service meets its specification - no more and no less. The implementation tools of section 4.3 support the implementation process.

4. *Testing* During the test phase, the implementation is certified to meet its specification. This certification may range from on the one hand common test practices relying on judgement and experience, to on the other hand a formal proof that the system as implemented possesses the desired properties. Friends currently develops a validation tool that checks whether test runs conform to the service specification (see section 4.2.2).

5. *Deployment* In this final phase, the deployment tools described in section 4.4 are used to deliver a deployable Friends service.

The phases, though presented in a linear way, will be performed in iteration. Notice, however, the explicit distinction between design and implementation activities. This is reflected in the tool architecture of the Friends service creation environment, as shown in Figure 2 (the right part of Figure 1 in detail).

4.1.3 Modelling library

Experience obtained with designing services leads to the recognition of re-usable components as well as *patterns* (sometimes called architectures). These components and patterns are stored in a modelling library. Testbed Studio supports the sharing of library elements (components) between several models.

Figure 5 shows an example: the ssUAP-USM-SSM pattern that has to be taken into account by each Friends service. To create different Friends services, the three basic components are extended with service specific components. Due to the used architecture the SSM is the logical component to implement the basic service management functions. The ssUAP components are extended with the graphical user interface elements, like e.g. the display of a video conferencing component or the interface of a shared whiteboard.

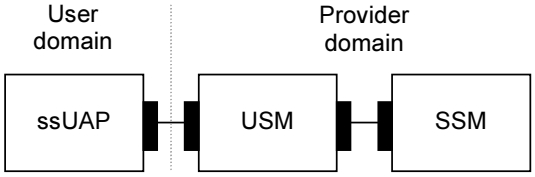

Figure 5. Friends service session pattern

4.2 Analysis tools

Analysis tools assist the service developer in analysing the properties of services as represented in service models *at design time*, and the properties of services as exhibited by service implementations *at run-time*.

4.2.1 Design time analysis

Testbed Studio includes tools supporting step-wise simulation and functional analysis of behaviour models.

Simulation. The complete specification of the behaviours of components allows one to simulate the service provided by individual components as well as the service provided by a composition of components. For example, simulation of the shared whiteboard behaviour model has shown that updates of the whiteboard may be lost in case of two simultaneous setBoard invocations by different ssUAPs. The reason is that an ssUAP may invoke a setBoard operation before the event notification of a previous setBoard from another ssUAP has been properly processed. The later setBoard simply overwrites the previous ones. Such design errors are typically detected during simulation. Figure 6 depicts the simulator control window.

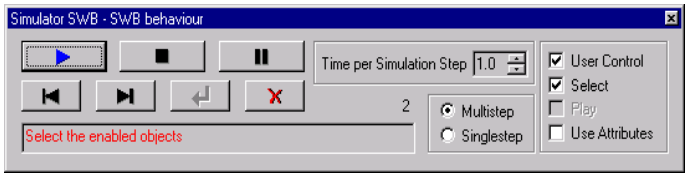

Figure 6. Simulator control window

Functional analysis. Testbed Studio enables a service designer to perform the following functional analyses on a behaviour model:

- *tracing*: checks whether a certain sequence of operations is always/ever/never executed;
- *liveness*: checks whether one, all or at least one operation invocation from a certain set of operations causes the invocation of at least one or all operations from another set;
- *combined occurrence*: checks whether the invocation of the operations from a certain set either exclude each other, or always/sometimes/never happen all together;

- *safety*: checks whether the invocation of each, all or at least one operation from a certain set requires the invocation of at least one or all operations from another set.

Verification. Based on the conceptual model underlying Amber, a general technique has been developed to enforce the correct replacement of an abstract behaviour by a more concrete behaviour, called *behaviour refinement*. A conformance relation defines which concrete behaviours are valid refinements (implementations) of the abstract behaviour, while it guarantees that the behaviour characteristics prescribed in the abstract behaviour are preserved by the concrete behaviour. For a further reading on this technique, we refer to [QuSF99]. Since this technique can in principle be automated, tools are planned that support conformance assessment after each design step.

4.2.2 Run-time analysis

Besides the common debug facilities provided by (Java) implementation environments, the Friends service creation environment adds two powerful techniques to test the actual implementation of a Friends service at run-time.

Monitoring. A monitoring framework is developed that enables the monitoring of interactions between distributed components [DBZS00]. Components interact by invoking operations on each other via a CORBA-compliant middleware platform. Operations can be synchronous (called interrogations), in which case a result is returned, or asynchronous (called notifications), in case no result is returned. An operation invocation (and the return of its result) involves the transmission of a so-called request object between the invoking and invoked component. The request object contains the operation name, operation arguments, results and other (e.g., context) information. Figure 7 depicts the monitoring points of a synchronous operation.

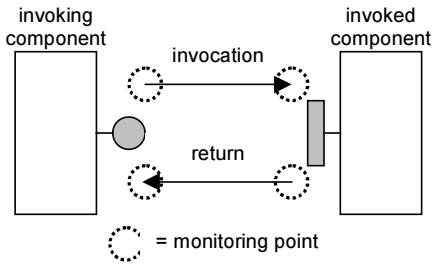

Figure 7. Monitoring points

When the operation is being executed, each monitoring point produces a so-called *interaction event* that is recorded. In this way, the monitoring framework is able to monitor the (real-)time ordering of interaction events. The interaction events are graphically represented using Message Sequence Charts (MSCs). Figure 8 depicts an MSC representing an execution of the SWB service.

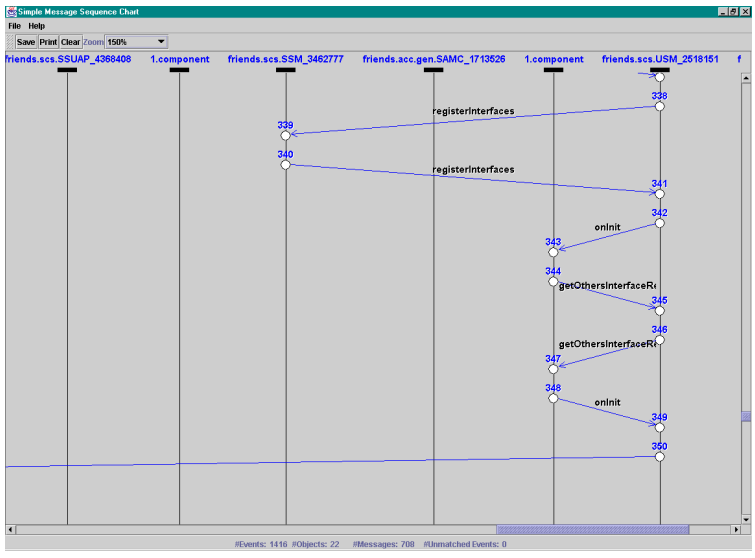

Figure 8. Message sequence chart

The monitoring framework is also able to reconstruct the causal relationship between operations. For this purpose, monitoring information is sent between components using the context field of the CORBA request object, and is propagated through components by tagging threads of execution that process the operation.

A prototype of the monitoring framework exists. The monitoring framework is integrated into the component framework, such that the application (service) developer is not burdened with monitoring issues. Currently, the monitoring framework is developed further to support different types of events, e.g., life-cylce events, QoS events and user-defined events, which

can be used for different purposes, such as accounting, load-balancing, QoS control and testing.

Conformance testing. Based on the MSCs obtained using the monitoring framework, a technique for conformance testing is developed. This technique allows a service developer to check whether individual runs (executions) of a service implementation conforms to an Amber model of the service, which either represents the external specification of the service or one of its designs. The following steps are distinguished (see Figure 9):

1. *monitoring*: a single run of the service implementation is monitored using the monitoring framework described above. The obtained MSC represents a partial order of interaction events, which is called a *real trace*;
2. *mapping*: the real trace is transformed into an *abstract trace* that can be compared to the Amber model. This transformation involves, amongst others, the abstraction (removal) of operations that where not considered yet at modelling level. Furthermore, differences between naming conventions used at modelling and implementation level may have to be resolved, in case the generation tools as explained in section 4.3.1 and 4.3.2 were not used;

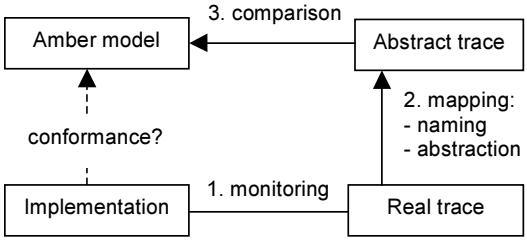

Figure 9. Conformance testing steps

3. *comparison*: the abstract trace is compared with the Amber model, using the simulator of Testbed Studio. Successive operation invocations defined by the abstract trace should also be allowed by a step-wise simulation of the Amber model.

A tool supporting these steps has been developed. The kind of automated support provided in the mapping step strongly depends on the assumptions that can be made on the type of refinements made during the design process. This will be elaborated in a forthcoming paper dedicated to this conformancing testing technique.

4.3 Implementation tools

Implementation tools assist the service developer in implementing (compositions of) components that provide the requested services.

4.3.1 IDL and CS generation

The 'final service design' specifies a Friends service in terms of an assembly of components. Some components may already be available, others may be missing. The latter components need to be implemented. The first step in the implementation process is a (black-box) specification of the functionality of the component. Such a specification is already part of the final service design.

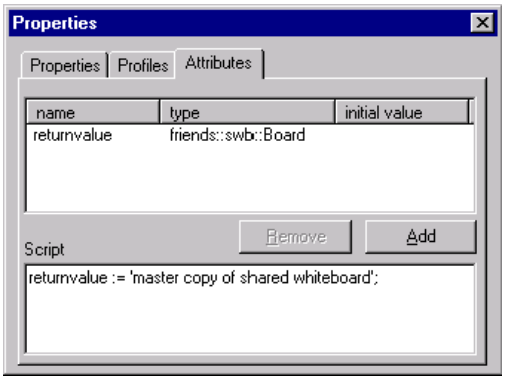

Figure 10. Interaction parameters

Referring to the so-called Service Session Manager component of Figure 3 and 4 (SWB-SSM), the interface i_SwbServer provides two operations, called setBoard and getBoard. Each operation is specified as a *behaviour block* containing an invocation and a return *interaction*. Additionally, an exception interaction may be specified (not shown in the figure). For each interaction parameters can be specified, as shown in Figure 10 for the interaction getBoard return. The parameters of an invocation interaction and a return interaction model the input and output parameters of the corresponding operation, respectively. Modelling interactions and their parameters in this way, interface descriptions (CORBA IDL) and 'component specifications' (CS; description

of sub-components and the relation between invoking and invoked interfaces) can be generated. This functionality has been implemented, including the reverse engineering. Figure 11 shows the resulting IDL for the SWB-SSM component.

```
// This file was generated by IDLGen
// Source file: D:\friends\src\friends\tools\IDLGen\SWB.xmb
// Date: 13-mrt-00 12:18:27

#ifndef SWB-SSM_IDL
#define SWB-SSM_IDL

// IDL file for component SWB-SSM
module friends {
        module swb {
                struct i_SwbServer {
                        string itfType;
                };
                struct Board {
                        integer version;
                };
        }; //swb
}; //friends

module SWB-SSM {
        interface i_SwbServer {
                void setBoard(in friends::swb::Board board);
                // invoke: board := 'setBoard invocation'.board;
                // 'master copy of shared whiteboard'.update(board);
                friends::swb::Board getBoard();
                // return: returnvalue := 'master copy of shared whiteboard';
        }; //i_SwbServer
}; //SWB-SSM

#endif
```

Figure 11. Generated IDL specification of the SWB-SSM.

4.3.2 Code template generation

IDL and CS specifications can in turn be used to generate stubs or skeletons, as a next step in component implementation. For this purpose, Friends (DSC) tools already exist as described by Batteram et al. [BBVD99]. In this stage, CORBA Component Descriptors (CCD, as defined by the CORBA Component Model [CCM99]) are generated as well.

4.3.3 Component implementation (atomic components)

Standard (Java) development environments are used to implement components. Notice that interface descriptions (IDL) are part of the specification of the component, but are not enough. To support a proper implementation, behavioural specifications are required as well. The AMBER specifications (as a communication means) can be used for this purpose. In current practice, message sequence diagrams are commonly used to support component design. These message sequences are often made by hand. However, message sequences can be derived from the service specification of a component. A tool has been implemented to generate message sequence diagrams from AMBER specifications. The step-wise simulator of Testbed Studio generates a trace from a model (already implemented), and converts this trace to standardised message sequence formats to be visualised.

4.3.4 Configuration (SCGen)

Functionality related to, e.g., access control, authorisation, accounting and service invocation are identical for each Friends service. Aiming at service creation, i.e., the fast and flexible creation of new service running on this platform, one wishes to abstract from such generic functionality and to focus on the service-specific functionality only. The integration of generic functionality should be handled automatically, or at least not being the concern of the service developer.

A tool has been built, called SCGen, that automatically relates the Friends services to already available generic functionality. The tool is described by Verhoosel et al. [VWBB99]. As shown in Figure 12, all kind of parameter options can be specified in AMBER models (in so-called *profiles* of an *entity*), whereupon the code generator SCGen uses this information.

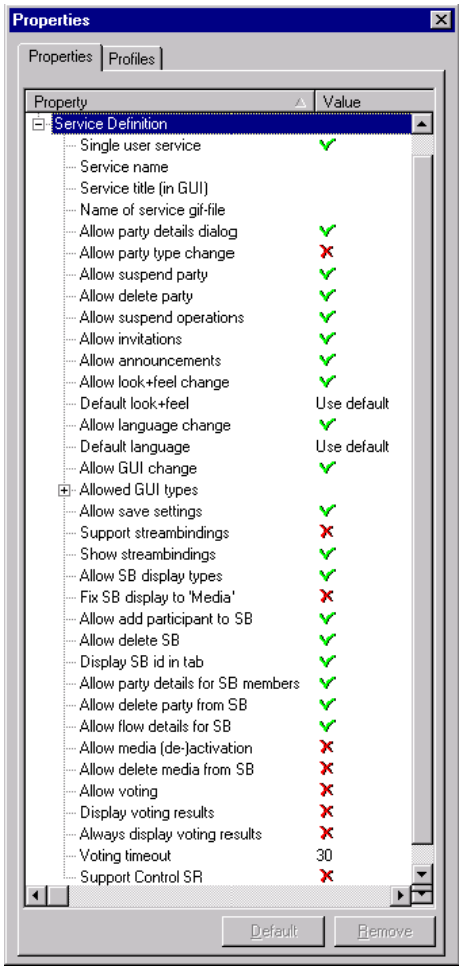

Figure 12. Configuration parameters for generic service functionality

4.3.5 Component repository

The component repository contains the (binary) components, which are stored as a zip-archive. Complying with the CORBA Component Model [CCM99], a Software Package Descriptor (SPD) is part of this archive.

4.4 Deployment tools

4.4.1 Assembly

Components are composed into larger ones, providing services to be deployed. In Friends, we migrated to the OMG CORBA Component Model (CCM) standard and the corresponding formats [CCM99, chapter 10]. CMM prescribes that for a compound component we not only need a CORBA Component Descriptor (see section 4.3.2), but also a Component Assembly Descriptor (CAD). This CAD can be generated from the AMBER models, and has been implemented.

4.4.2 Packaging

To be deployed, a component has to be packaged according to formats as required by the deployment platform. Identical to 'atomic' components, compound components are stored as a zip-archive as described in section 4.3.5. Tools for automated packaging still have to be implemented. The CORBA Component Model allows the inclusion of (formal) behaviour descriptions in the software package, which we obviously intend to do.

5. Conclusions

In this paper, we sketched our ideas about model-based service creation and illustrated them in the context of the Friends project. Many of these ideas have already been implemented in the Friends platform, such as a graphical language for high-level component assembly, code generation from component specifications, and analysis tools. Through implementation of the service creation environment, we aim at proving the applicability of our approach, and in particular the added value of behaviour specifications.

Other ideas need further elaboration before they can be implemented. Some short term research issues are the extension of AMBER with concepts tailored to service creation, parameterisation of model-components to support model-based customisation, and searching components based on functionality. Research issues on the long(er) term are mainly in the area of analysis techniques, such as improved model checking support, verification techniques to assess the conformance between service designs defined at successive abstraction levels, and the modelling and analysis of performance aspects.

Concluding, we believe that our research contributes to the goal of developing a platform supporting rapid and correct service creation. In particular, the application of high-level specification languages and supporting (analysis) tools should enable the service developer to build a service from software components that can be considered at high(er) abstraction levels, and to assess the correctness of these components and their composition.

References

[BaBa98] Bakker, J.-L., and H.J. Batteram, Design and evaluation of the Distributed Software Component framework for distributed communication architectures, In: *Proc. Of the 2nd Int. Workshop on Enterprise Distributed Object Computing (EDOC'98),* November 1998, p. 282-288.

[BBVD99] Batteram, H.J., J.-L. Bakker, J.P.C. Verhoosel and N.K. Diakov, 'Design and implementation of the MESH service platform', In: Proceedings of TINA'99 Telecommunications Information Networking Architecture Conference, Oahu, Hawaii, USA, 12-15 April 1999.

[Bern96] Bernstein, P.A., "Middleware: A Model for Distributed Services." Communications of the ACM 39, 2, February 1996, pp. 86-97.

[BJPW99] Beugnard, A., J.-M. Jézéquel, N. Plouzeau and D. Watkins, 'Making components contract aware'. IEEE Computer (July 1999), p. 38-45.

[BoLV95] Bolognesi, T., J. van der Lagemaat and C.A. Vissers (eds.), *LOTOSphere: Software development with LOTOS*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1995.

[CiRo99] Cicalese, C.D.T. and S. Rotenstreich, 'Behavioral specification of distributed software component interfaces'. IEEE Computer (July 1999), p. 46-53.

[CCM99] CORBA Components - Volume I, OMG TC Document orbos/99-07-01, August 2, 1999, http://www.omg.org/docs/orbos/99-07-01.pdf.

[Diak99] Diakov, N., *Survey on products for visual assembly tools for Java*, Amidst document WP1/N007, University of Twente, Enschede, The Netherlands, 3 June 1999, http://amidst.ctit.utwente.nl/workpackages/wp1/documents/wp1n007v01.pdf

[DBZS00] Diakov, N.K., H.J. Batteram, H. Zandbelt and M.J. van Sinderen, 'Monitoring of distributed component interactions'. In *Proceedings of 7th Int. Conf. on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS 2000) in Enschede, The Netherlands, October 2000*.

[EJO+99] Eertink, H., W.P.M. Janssen, P.H.W.M Oude Luttighuis, W.B. Teeuw, C.A. Vissers, 'A Business Process Design language', Proceedings World Congress on Formal Methods (FM'99), 1999.

[FRIENDS] http://friends.gigaport.nl/; http://friends.telin.nl/

[FrJa98] H.M. Franken and W. Janssen, 'Get a grip on changing business processes Results from the Testbed-project', *Knowledge & Process Management* (Wiley), vol. 5, no. 4, December 1998, p. 208-215.

[GHJV95] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "*Design patterns : elements of reusable object-oriented software*". Addison-Wesley, Reading, MA, 1995.

[Holz97] Holzmann, G.J., 'The model checker SPIN'. *IEEE Trans. on Soft. Eng.*, vol. 23, no. 5, May 1997.

[QFS+97] Quartel, D.A.C., L. Ferreira Pires, M.J. van Sinderen, H.M. Franken and C.A. Vissers, "On the role of basic design concepts in behaviour structuring", *Computer Networks and ISDN Systems*, vol. 29, no. 4, March 1997, 413-436.

[QuSF99] Quartel, D.A.C., M.J. van Sinderen, and L. Ferreira Pires, "A model-based approach to service creation", In: *Proceedings of the Seventh IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, IEEE Computer Society, 1999, 102-110.

[QuTe00] Quartel, D.A.C., and W.B. Teeuw (Ed.), *Modelling FRIENDS components in the language AMBER*, Report FRIENDS/WP3/N012/V01, Telematics Institute, Enschede, The Netherlands, 21 Februari 2000, https://extranet.telin.nl/docuserver/dscgi/ds.py/View/Collection-745.

[ReMa97] Rechtin, E. and M.W. Maier, *The art of systems architecting*. CRC Press, Boca Raton, FL, 1997.

[Szyp98] C. Szyperski, Component Software, Beyond Object-Oriented Programming, Addison-Wesley, ACM Press, New York, 1998.

[TINA97] Kristiansen, L. (ed.), *Service Architecture: Version 5.0*. TINA-Consortium, June 1997. http://www.tinac.com/specifications/specifications.htm

[UML] http://www.rational.com/uml/resources/documentation/

[VWBB99] Verhoosel, J.P.C., M. Wibbels, H.J. Batteram and J.-L. Bakker, 'Rapid service development on a TINA-based service deployment platform'. In: Proceedings of TINA'99 Telecommunications Information Networking Architecture Conference, Oahu, Hawaii, USA, 12-15 April 1999.